DOCUMENT RESUME

ED 263 206 TM 850 701

AUTHOR Psotka, Joseph
TITLE Reflections on Computers and Metacognition.
PUB DATE [85]
NOTE 10p.
PUB TYPE Viewpoints (120)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS *Artificial Intelligence; *Cognitive Processes; Cognitive Restructuring; Computer Assisted Instruction; *Computer Science; Elementary Secondary Education; Heuristics; *Learning Processes; *Metacognition; Problem Solving; *Programing Languages
IDENTIFIERS *Debugging (Computers); LISP Programing Language; LOGO Programing Language; Process Product Relationship; Process Product Research

ABSTRACT

Current notions of metacognition merge with the predominant scientific model used in psychology, that of information processing. Metacognition is seen as a control process that governs the action of more elemental cognitive skills. Given the centrality of this notion, it is important that metacognition should be examined in detail. From the point of view of how metacognition relates to highly sophisticated computer systems in use today and in development for the future the following are discussed: (1) debugging and metacognition; (2) cognitive compatibility and learning; (3) problem definition and analyses; (4) procedures and facts; (5) process and product; and (6) tools for the mind. The perspective of this paper is that higher order languages like logo and lisp are only beginning to have an impact on the understanding of psychological processes. The whole development of computer-aided instruction, artificial intelligence, and special environments for instruction will unquestionably have profound effects on education, but its more enduring and important effect may well be increasing the understanding of cognition and learning. (PN)

Reflections on Computers and Metacognition

Joseph Psotka
Army Research Institute
5001 Eisenhower Avenue
Alexandria, Virginia 22333-5600

REFLECTIONS ON COMPUTERS AND METACOGNITION

Joseph Psotka
Army Research Institute
5001 Eisenhower Avenue
Alexandria, Virginia 22333-5600

This paper is largely in response to a host of empirical data set out carefully and in great detail in a group of papers presented at AERA (Campbell et al., 1985; Clements, 1985; Dytman et al., 1985; Miller et al., 1985; and Zelman, 1985). It also reflects some of the discussion at that session "On using Logo with students." and later with Wallace Feurzeig. The papers presented at the session covered many topics from several perspectives, but recurrent in their themes was the importance of Logo for metacognitive growth. Current notions of metacognition merge nicely with the predominant scientific model used in psychology these days, that of information processing. Metacognition is seen as a control process that governs the action of more elemental cognitive skills. Given the centrality of this notion, it seems timely and important to examine it in greater detail on its own, and especially from the point of view of how it relates to highly sophisticated computer systems in use today and in development for the future.

The perspective of this paper is that, in fact, higher order languages like logo and lisp are only beginning to have an impact on our understanding of psychological processes. These languages are still quite crude tools for exploring the potential interactions between computers and people. They are nevertheless very rich tools for exploration, but insight into the process suggests that these languages need to be made replete with specialized tools to further their effects. We are only beginning to see the development of these kinds of specialized systems that could very much help instructors improve their using logo with students. The whole development with computer-aided instruction, artificial intelligence, and special environments for instruction will unquestionably have profound effects on education, but its most enduring and important effect may well be increasing our understanding of cognition and learning.

DEBUGGING AND METACOGNITION

Perhaps no other concept derived from computer science has captured the imagination of psychologists so powerfully in their attempt to explain instruction than the notion of "debugging" (c.f. Burton, 1982). Superficially, it seems such an appropriate expression of removing students' errors in thinking. Let the program run, and where it breaks, remove the offending little piece of code. With simple programs, the task really is simple.

With sophisticated programs of any respectable length, the task is not only not so simple, it begins to take on much more profound characteristics to make it look more like "learning" than just error correction. In more complex programs, not only does one have to find the bug, but one has to restructure the knowledge (data and procedures) in the program to make it behave properly. This turns out to be quite complicated in many ways.

The more fully developed notions of "debugging" turn out to be central to both learning and computer programming. Within the learning context, analyses of childrens' learning has often taken the approach of analysing arithmetic and algebra. This has several distinct advantages, because both subject matters are heavily algorithmic and allow themselves to be modelled on computers relatively easily (compared to the poorly understood skills of civics, for instance). Several intensive investigations (Brown and Burton, 1978; Sleeman, 1982) have produced models of student learning in which errors are realistically described not by missing knowledge or facts, but by the imaginative creation of new rules: "buggy" rules or "mal" rules, as they have been called. A mammoth theoretical undertaking has forged a link between these buggy rules and the source of instruction: textbooks and the order of learning exercises (Van Lehn, 1983). This evidence forces a conception of the learner as an active theory-builder whose rules are organized into theoretical statements that try to describe the world as it is known.

The buggy rules are theoretically coequal with other rules and facts in any student's understanding of the world, and in this respect, can be thought of as analogous to the individual statements or function definitions in any programming system. In a programming environment where one is editing functions not only interactively, but in a structure editor that is making changes as they are seen, debugging becomes very complicated. The very act of looking at a statement definition may change it directly or have some indirect effects on other parts of the environment. In order to debug in such an environment, the proper metaphor seems to be one of "tiptoeing" through one's knowledge structures, carefully trying not to destroy what one steps on. Looking at this process more closely, as a metaphor, it would seem that there are two key elements of the process: finding a place to stand; and having a control system that interprets what one finds and decides what to change. Given this metaphor, debugging is very closely allied to self-reflection, since being able to examine clearly what one knows (the existing code) is a precondition for further learning and restructuring.

What sort of computer system could maintain this sort of self-reflection. Clearly, it does not yet exist (although there are some intimations in the work of Smith, 1985). Two simple processes come to mind. In one, the computer takes a snapshot of its entire memory every fixed moment of time and stores it away for future reference. In the other, there are more than one computer (in fact there are as many as may be needed) and they are each doing the same thing, but time-lagged by some fixed or variable amount. When one wants to know what it was doing a

few moments ago, it calls on the appropriate computer. In both cases, an interpreter that can examine the past state of knowledge and decide on what to change so that a "better" present may come about is still needed (and undefined). Clearly, these are very wasteful and inefficient models of self-reflection, but they may at least provide some sense of the task requirements for this complex ability. Just how much of this capability really will be required remains to be determined, empirically and theoretically.

Let us try to create some notion of the human context for this discussion: It seems that we are able to reflect quite deeply on our past actions and decisions. We can, for instance, retrace in our minds the course of an argument or conversation with someone and realize errors or shortcomings in our logic. We can, as another example, learn from falling off a bicycle something about how not to maintain balance, and this something may be quite complex and difficult to articulate. The point to be drawn from these examples is that we must maintain some relatively detailed description of our states of knowledge at particular times and make use of these descriptions to alter those very states of knowledge through some sort of process of self-reflection. Now the artificial self-reflective systems outlined earlier provide a ground for creating this self-reflection, at the very least, and they may begin to suggest some of the mechanisms or procedures for simulating this fundamental human capacity.

What one needs to make use of these self-reflective systems is for them to be aware of their reflectivity and capable of using this knowledge. They need rules or heuristics for comparing their present state of knowledge with the past so that they can find the appropriate past state to modify. What sort of comparison could this be? For faulty deductions one might use appropriate backtracking mechanisms. But what does one do about falling off a bicycle? Well, again one could backtrack to the point where falling began and then try to do things differently: e.g. "Perhaps I turned the wheel too sharply. Maybe next time I should shift my weight to counterbalance, ... etc." It may be that we will not begin to understand the utility of these systems as they currently exist in our minds until we begin to explore them on the simulacrum of the computer.

COGNITIVE COMPATIBILITY AND LEARNING

If the experiences with computers are to be useful for theorizing and for learning, the activities with symbols must correspond in some simple way with the concepts and ideas in the mind. In the main, the discussion about self-reflection was really premised on this notion: that we could learn from the computer metaphor only insofar as it corresponded with what we know to be true about human self-reflection. This should have an implication for using Logo with students for learning. Symbolic interaction with Logo should affect the complex development of mental life to the degree that activity is cognitively compatible or has some of the

other characteristics that good human interface designers have been ardently promoting: direct manipulation and WYSIWYG (What You See Is What You Get!).

Logo has not been designed directly for this effect, but it seems to share some of these primal characteristics. It is a clean, hierarchical, procedural language first developed in 1966 at BBN as an offshoot of Lisp. It was intended to provide a principled conceptual framework for the exploration of mathematics instruction. It has turned out to be a very valuable environment for exploring human thinking skills. However, it is still pretty much in its raw, undeveloped state. Given this state, it seems silly to expect it to have dramatic effects on children's thinking without further development. As an environment for discovery, it certainly provides rich possibilities, but these have been investigated only with the Logo turtle. What is clearly needed are other "turtle-like" environments that encourage one to explore areas of knowledge like music, language, physics (c.f. DiSessa, 1984), art, and anything else one would like to study from kindergarten to post-graduate activities. Strangely, the business world with its Visicalcs and MacPaints may provide the best examples of these sort of environments: highly sophisticated, special purpose programs that allow a great degree of freedom in their use but provide special assists that make performing certain acts very easy and transparent. If logo is to survive and grow, it will need these sort of environments for instruction.

One of the strengths of these kinds of tools is that they build on existing knowledge derived from real life experiences. That is certainly true of the logo turtle. The more a computer environment recreates the real world, the easier it is to operate, and the more creatively one may be able to use it. The logo turtle encourages children and other users to imagine themselves in the computer environment and bring their knowledge and experience of the world into that environment. This may build on a strength we already have, namely the perceptual systems' ingrained tendency to recognize relationships on the basis of real world statistics (Egon Brunswik called it "ecological perception"). It should not be surprising then that (as Campbell et al. found) children tend not to use the command to tell the turtle to go backwards. After all, how often do we or children walk backwards to get something? Abstracting the special capabilities of the turtle beyond what we would physically do in that environment is the critical step. How to encourage that sort of abstraction is what educators and psychologists need to learn.

PROBLEM DEFINITION AND ANALYSIS

It is no secret that an essential component of clear thinking is to be able to divide large or poorly defined problems into manageable chunks --- mind-sized bites --- and order them rationally. Logo (as any higher order language) forces people to decompose problems into discrete components. The simple but enormous advantage the computer environment then provides is that these components are

"runnable": input, transformations, and output are clearly specifiable and can be made directly inspectable. The system can give immediate feedback on the quality of the decomposition. It can make obvious missing components. It can make visible the intermediate products that may be essential for suggesting the next step. In a complex environment, it can even provide the elemental support of existing modules that may be incrementally modified to perform the needed activities.

A modular, hierarchical environment can provide much support to facilitate crisp logical thinking, but it needs specialized tools to adequately perform this function. By and large, these tools are not yet available in Logo, even though some things (such as idea processors that help organize and restructure large bodies of notes) are beginning to be available in business and the AI community (cf. Halasz and Moran, 1984).

One of the real strengths of Logo has been its support of a graphics environment with turtles and sprites. The graphics not only support debugging facilities for symbolic programs and a wonderful student interface; they also let analogical mental structures find direct visible support. They create a bridge between the symbol systems of the mind and the computer. In a very real sense a graphics support environment can become a physical symbol system in which mental objects are directly inspectable. With any luck, this graphics support can let mental structures grow with greater power and flexibility to cut apart problems, reorient them, and suture them together in novel and creative ways.

There is a natural precedent for this speculation that the computer can mirror the mind and facilitate the growth of new symbol systems. Olson (1985) has pointed out that the growth of printing brought about a parallel growth in language. A prominent feature of this change was an increase in the number of state descriptors (e.g. assert, state, aver, affirm, etc.) since the printed word had to carry the surplus meaning of gesture and facial expression. These words found their way into regular discourse as well.

Logo is a communication system as well. The words and ideas used to communicate with the computer may well be turned on one's own knowledge to guide the formulation of modular, structured components. It is naive to believe that we have privileged access to our own ideas. A computer environment may in fact be able to reify those ideas so that they may be more readily inspected and analysed.

PROCEDURES AND FACTS

Debugging one's own knowledge first assumes an inspection system that can safely examine that knowledge without interfering with it. It is a convention these days to separate knowledge into two kinds: procedures and facts. Procedures are the mind's actions. They are intuitive because we have access to them only by

running them, or in the form of vague kinaesthetic imagery. Facts are retrievable in the form of words, images, or other conscious descriptors.

The distinction bwteen procedures and facts in mental life follows a similar distinction between functions and data in computer life. There is a continuing controversy over the relative merits of procedural versus data-oriented programming that is epitomized by the differences between Lisp and Prolog. It is clear that each has its merits but that these are still developing and presently poorly defined.

Fundamentally, an entity or symbol in a machine may be regarded as either a datum or as part of a procedure, depending on the context and purpose. In fact, there may be a kind of Heisenberg Principle of Uncertainty involved with debugging these symbols. One may be able to examine the symbol either by running it as part of a procedure or by inspecting its values directly. In a complex system, it may not be possible to do both. Either action may have unintended side-effects: in a complex system many things may change without leaving a trace, so that the system is in effect non-deterministic. The procedures may be viewed as plans with default data values bound to the components of the plans. If one executes those plans to see where they lead, this very process may destroy the default values that existed at the time. Furthermore, at any given moment the actions of the procedures depend on the conditions of the data value in intricate, interactive ways. Changing the bindings of the data intimately affects how the procedures run. A debugging environment that chooses to debug by running procedures risks destroying the values: and a debugging environment that chooses to reset values runs the risk of deactivating certain components of the procedures.

As Dytman et al. have shown, children's debugging activities reflect the operation of this uncertainty principle. Children try to solve their problems by systematically (and unsystematically) 1.) Changing data values, and 2.) changing lines in their procedures. Usually they focus on one or the other at a time. Often they must begin over again when both are tried and the environment is destroyed.

PROCESS AND PRODUCT

In an educational environment, products are often of secondary importance: the process of arriving at the right solution is the real goal, and the particular exercise of that process (while it must be selected with care to foster learning expeditiously) only serves that goal. In her analysis of programming activity, Dytman et al. find that the relative experts ask about procedures, while novices ask about outcomes and products. These young experts appear to be able to "run" the procedures in their minds to envision their outcomes. The others still need reassurance about what the real outcome is supposed to be. They still have to learn to adapt their self-inspection schemes and develop ways to run their internal plans to determine the proper outcome.

It is interesting to speculate how this resource-seeking activity of the young changes in older children. Zelman makes a well-documented case that adolescents learning to program in Logo generally are quite willing to take procedures provided from teachers and adopt them as their own. By changing some of the data values, the students can produce their own unique products. This approach of adopting procedures seems consistent with general educational principles where teachers try to instil procedures and provide rote instructions for following them; but it seems inappropriate and inconsistent with accepted software practices where the program is the important product, not the actual output of the program. It may be that teaching programming skills raises new problems of plagiarism and independence of effort.

TOOLS FOR THE MIND

Given the existence of these realistic and objective examinations of Logo's effectiveness and potential uses, it seems time to call for a public examination of the directions languages like Logo should take. It seems clear that there is now a need for more than just an open-ended exploratory environment. There is a real need for a tool for self-experimentation and revision; to support the externalization of our thought processes. Miller's finding that a software product for building IQ is relatively effective in many ways, provides one direction for a set of environment tools. But clearly, it needs the added flexibility of being adaptible to each student's own purposes.

Zelman's study provides uneqivocal evidence for the importance of structured software and guidance within the Logo environment. There is a real need for curriculum related materials and exploratory environments in the same system. Lisp machines coming out of the AI community provide an outstanding model for the kinds of facilities that are needed. Above all, they have emphasized the complete integration of the computing environment: everything is available at the same time and roughly with the same priority status.

As a final speculation for future directions, think about a system where everything is done in dynamic memory. Think about an integrated environment in which every change affects how everything else works. And then consider that you cannot turn the machine off to reset all its values. Think how important debugging and knowledge restructuring then becomes. And consider how much we might learn about the human mind with such a model.

REFERENCES

Brown, J. S. and Burton, R. R. Diagnostic models for procedural bugs in basic mathematical skills. Cognitive Science, 1978, 2, 155-192.

Burton, R. R. Diagnosing bugs in a simple procedural skill. In D. Sleeman and J. S. Brown (Eds.), Intelligent Tutoring Systems, London: Academic Press, 1982.

Campbell, P. F., Fein, G. G., Scholnick, E. K., Frank, R. E., and Schwartz, S. S. Initial mastery of the syntax and semantics of Logo. Chicago: AERA, 1985.

Clements, D. H. Effects of Logo programming on cognition, metacognition, and achievement.Chicago: AERA, 1985.

DiSessa, A. A. A principled design for an integrated computational environment. Paper in preparation, 1984.

Dytman, J., and Wang, M. C. An investigation of the role of the learner in Logo learning environments. Chicago: AERA, 1985.

Feurzeig, W. Personal Discussions. Chicago: AERA, 1985.

Halasz, F. G. and Moran, T. P. Notecards. Personal communication.

Olson, D. R. Computers as tools of the intellect. Educational Researcher, 1985, 14, pp. 5 - 8.

Miller, G. E., Emihovich, C., Clare, V., and Froning, D. The effects of interactive programming on preschool children's self-monitoring. Chicago: AERA, 1985.

Sleeman, D. H. Inferring (mal) rules from pupils' protocols. Proceedings of the European Conference on Artificial Intelligence, 1982, 160-164.

Smith, B. C. Reflection and semanitics in Lisp. CSLI Report No. CSLI-84-8, 1984.

Van Lehn, K. Felicity conditions for human skill acquisition: Validating an AI-based theory. Xerox Cognitive Instructional Sciences Report Number CIS-21; November, 1983.

Zelman, S. Individual differences and the computer learning environment: motivational constraints to learning Logo. Chicago: AERA, 1985.